下载APP

25 | 容量场景之二：缓存对性能会有什么样的影响？

2021-05-17 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼

时长 18:46 大小 17.20M

你好，我是高楼。

上节课，我们经历了三个阶段的分析优化，分别解决了在压力线程不变的情况下，TPS 随时间增加而增加的问题，还有数据库加索引的问题，以及 Kubernetes 调度不均衡的问题。最后，TPS 曲线看起来挺正常了，但是命运不会因为我努力了就会放过我。

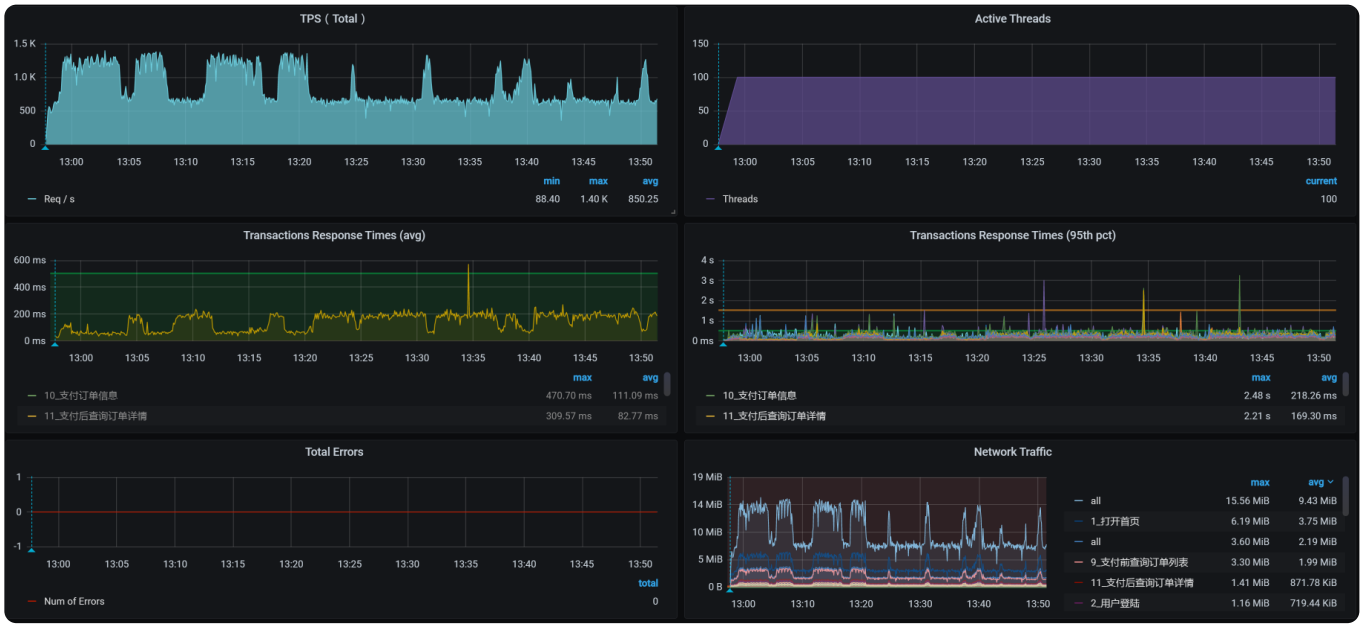
为什么这么说呢？因为在上节课中，我们的场景只持续了十几分钟，对于容量场景来说，时间还是不够长。你知道，压力持续十几分钟，且 TPS 显示正常，并不能说明系统没有问题。

因此，我又对系统进行持续的压力测试，就是在这个过程中，又遇到了新的问题……

第四阶段分析

场景压力数据

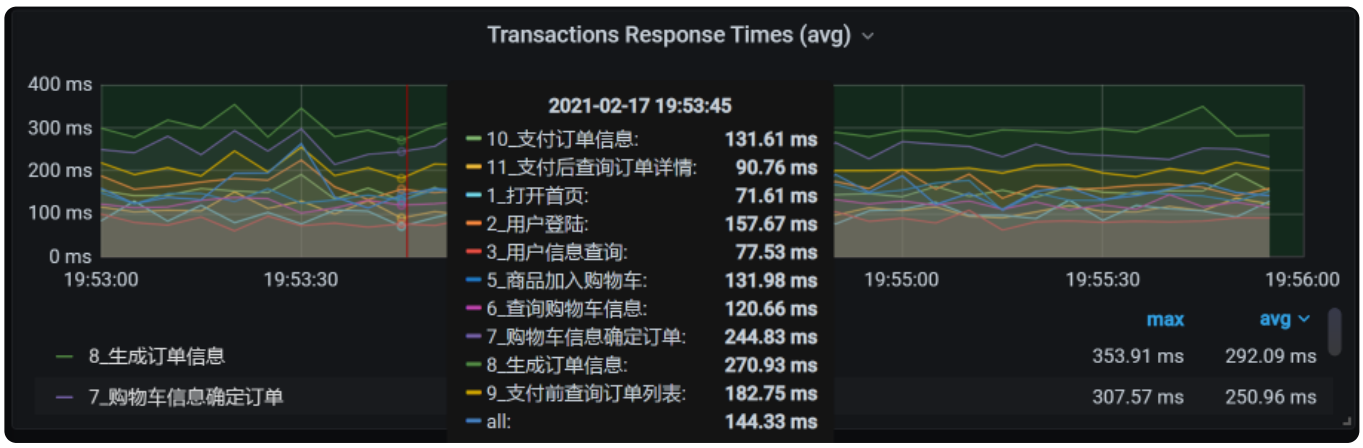
这是我在进行持续加压过程中，得到的场景数据：



看上面的曲线图就能知道，这是在压力持续的过程中，出现了 TPS 掉下来的问题，这是不能接受的。

拆分响应时间

针对上述问题，我们先来看一下现在的时间消耗。这是已经运行了一段时间的响应时间图：



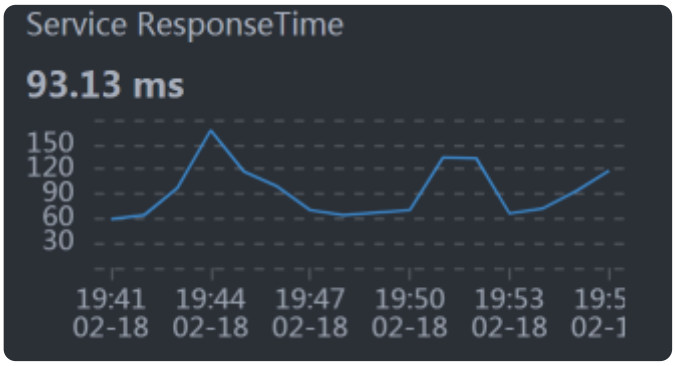
我们可以根据整体的平均响应时间，一个个分析这些接口的时间消耗在了哪里。其实，从这张图就能看出，所有的业务时间相比上一节课的响应时间图都增加了。由于所有业务的

响应时间都增加了，说明不是某个业务本身的问题，所以，我们任意分析一个接口就可以。

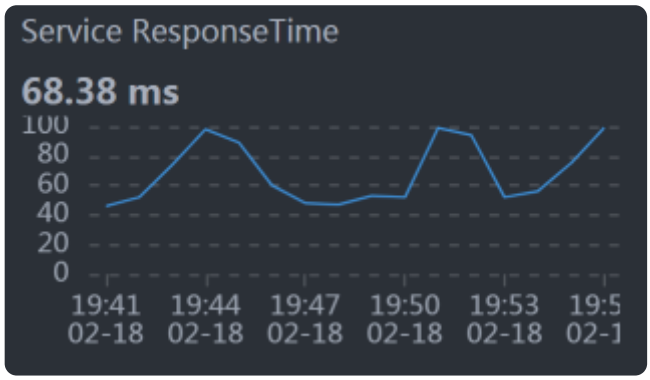
这里我用生成确认订单这个接口做时间拆分。在之前部署系统的时候，我们把 SkyWalking 采样率设置得非常低，只有 5% 左右，目的为了不让 APM 影响性能和网络。

下面这些数据是按分钟平均的。

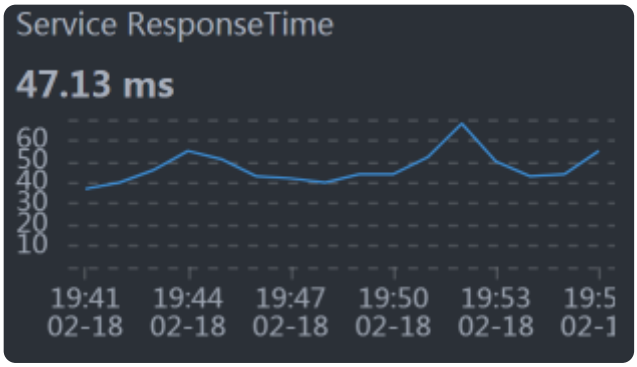
Gateway：



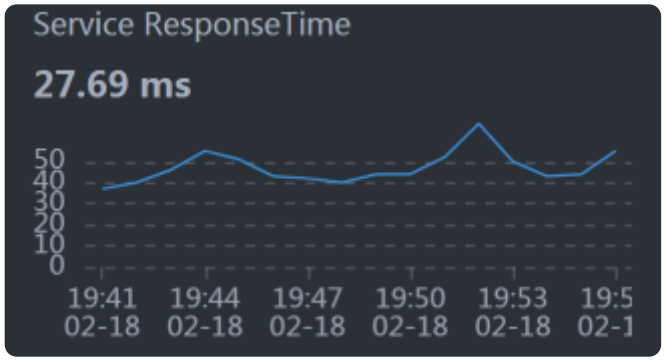
Order：



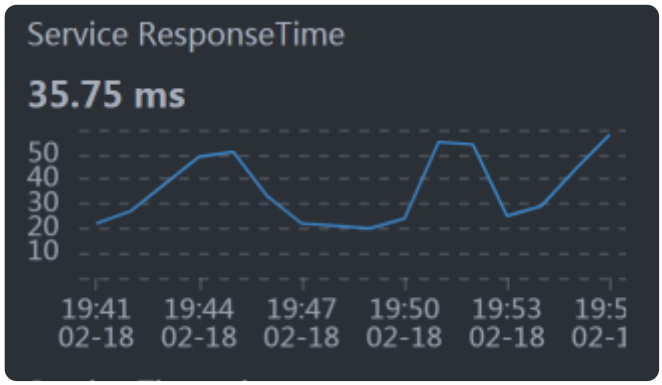
Cart：



Member：



Auth：



从数据上来看，似乎每个服务都和整体响应时间慢有关。悲催的场景总是这样。

不过，别慌张，我们仍然按照全局到定向的分析思路来走就可以了。我们先看全局监控数据。

全局监控分析

从全局监控的第一个界面来看，worker-4 上的 CPU 资源使用比较高，其次是 worker-6：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.84 week	7.64 GiB	4	1.06	13.09%	41.58%	29.95%	18.41 kB/s	97.34 kB/s	312	98	374.88 kbps	510.08 kbps
k8s-master-2	k8s-master-2	8.54 week	7.64 GiB	4	1.52	21.57%	23.67%	29.95%	0 B/s	93.69 kB/s	254	27	251.13 kbps	251.13 kbps
k8s-master-3	k8s-master-3	1.84 week	7.64 GiB	4	0.59	11.90%	36.03%	29.95%	13.81 kB/s	97.71 kB/s	309	103	309.14 kbps	578.34 kbps
k8s-worker-1	k8s-worker-1	2.54 hour	15.51 GiB	6	4.28	55.97%	34.63%	47.98%	3.57 MB/s	4.61 MB/s	30	90	56.86 Mbps	57.14 Mbps
k8s-worker-2	k8s-worker-2	2.54 hour	15.51 GiB	6	6.59	54.56%	27.41%	36.96%	23.02 kB/s	120.97 kB/s	27	721	21.62 Mbps	14.15 Mbps
k8s-worker-3	k8s-worker-3	2.54 hour	15.51 GiB	6	3.80	47.23%	24.95%	38.09%	47.44 kB/s	141.67 kB/s	28	501	23.73 Mbps	23.59 Mbps
k8s-worker-4	k8s-worker-4	2.54 hour	15.51 GiB	6	8.71	85.46%	51.62%	35.43%	98.07 kB/s	462.19 kB/s	35	25 K	37.28 Mbps	43.15 Mbps
k8s-worker-5	k8s-worker-5	7.45 week	15.51 GiB	6	3.44	42.63%	58.72%	38.43%	13.81 kB/s	10.03 kB/s	29	155	15.60 Mbps	14.59 Mbps
k8s-worker-6	k8s-worker-6	7.45 week	15.51 GiB	6	11.84	81.42%	34.40%	40.94%	9.21 kB/s	54.28 kB/s	47	361	28.68 Mbps	19.11 Mbps
k8s-worker-7	k8s-worker-7	7.15 week	15.51 GiB	4	4.71	53.65%	34.02%	42.34%	14.00 kB/s	53.30 kB/s	30	404	18.30 Mbps	13.59 Mbps
k8s-worker-8	k8s-worker-8	5.97 week	15.51 GiB	4	0.41	13.46%	33.74%	57.25%	0 B/s	4.85 kB/s	28	190	16.58 Mbps	16.67 Mbps
k8s-worker-9	k8s-worker-9	7.15 week	15.51 GiB	6	2.05	25.41%	28.44%	29.95%	18.41 kB/s	18.97 kB/s	37	78	27.59 Mbps	41.81 Mbps

我们一个一个来收拾。

我们先进入到 worker-4 中，执行 top/vmstat 命令，截取到一些重要的数据，下面这些是 worker-4 这个节点更为详细的全局监控数据：

复制代码

```
-- vmstat的数据
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu---
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
12  0      0 4484940   1100 4184984    0    0     0     0 45554 21961 58 25 14
 9  0      0 4484460   1100 4185512    0    0     0     0 45505 20851 60 25 14
16  0      0 4483872   1100 4186016    0    0     0     0 44729 20750 62 24 12
15  0      0 4470944   1100 4186476    0    0     0     0 45309 25481 62 24 13
14  0      0 4431336   1100 4186972    0    0     0     0 48380 31344 60 25 14
16  0      0 4422728   1100 4187524    0    0     0     0 46735 27081 64 24 12
17  0      0 4412468   1100 4188004    0    0     0     0 45928 23809 60 25 13
22  0      0 4431204   1100 4188312    0    0     0     0 46013 24588 62 23 13
12  0      0 4411116   1100 4188784    0    0     0     0 49371 34817 59 24 15
16  1      0 4406048   1100 4189016    0    0     0     0 44410 21650 66 23 10
..................


--- top的数据
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  935 root      20   0 6817896   1.3g  16388 S 301.7  8.7  71:26.27 java -Dapp
 1009 101       20   0   13500   2980    632 R  37.6  0.0  10:13.51 nginx: wor
 1007 101       20   0   13236   2764    632 R  20.8  0.0   3:17.14 nginx: wor
 7690 root      20   0 3272448   3.0g   1796 S  14.2 19.1   2:58.31 redis-serv
 6545 101       20   0  737896  48804  12640 S  13.9  0.3  12:36.09 /nginx-ing
 1108 root      20   0 1423104 106236  29252 S  12.2  0.7  16:28.02 /usr/bin/d
 1008 101       20   0   13236   2760    628 S   6.9  0.0   0:46.30 nginx: wor
 6526 root      20   0  109096   8412   2856 S   6.3  0.1   7:30.98 containerd
 1082 root      20   0 3157420 116036  36328 S   5.3  0.7  11:15.65 /usr/bin/k
 6319 nfsnobo+  20   0  759868  53880  18896 S   3.0  0.3   3:18.33 grafana-se
 6806 root      20   0 1632160  47276  17108 S   2.6  0.3   5:09.43 calico-nod
    6 root      20   0       0      0      0 S   1.0  0.0   0:14.19 [ksoftirqd
..................
```

从 vmstat 的 in 列可以看出，系统的中断数比较高；从 vmstat 中的 cs（context switch）来看，CS 也达到了 3 万左右。同时，sy cpu（syscall 消耗的 CPU）也占到了 25% 左右。这就说明我们需要着重关注 syscall（系统调用）层面。

定向监控分析

我们先查看中断数据，下面这张图就是全部的中断数据截图：

```
Every 5.0s: cat /proc/interrupts

           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
  0:        182          0          0          0          0          0   IO-APIC-edge      timer
  1:         11          0          0          0          0         89   IO-APIC-edge      i8042
  4:       1661          0          0          0          0          0   IO-APIC-edge      serial
  6:          3          0          0          0          0          0   IO-APIC-edge      floppy
  8:          0          0          0          0          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0          0          0   IO-APIC-fasteoi   acpi
 10:         29          0          0          0          0         27   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2, virtio3
 11:         76          0         22         35         52         42   IO-APIC-fasteoi   uhci_hcd:usb3, uhci_hcd:usb4, qxl
 12:         15          0          0          0          0          0   IO-APIC-edge      i8042
 14:          0          0          0          0          0          0   IO-APIC-edge      ata_piix
 15:          0          0          0          0          0          0   IO-APIC-edge      ata_piix
 24:          0          0          0          0          0          0   PCI-MSI-edge      virtio2-config
 25:      10300       1425       4780      18171      22852      25124   PCI-MSI-edge      virtio2-req.0
 26:          0          0          0          0          0          0   PCI-MSI-edge      virtio1-config
 27:         12       1071          0          0          0          0   PCI-MSI-edge      virtio1-virtqueues
 28:          0          0          0          0          0          0   PCI-MSI-edge      virtio0-config
 29:        107       4351    3083295         20          0          0   PCI-MSI-edge      virtio0-input.0
 30:          5          3          4          4         25         14   PCI-MSI-edge      virtio0-output.0
 31:         94    2023119         78    1297126          0        258   PCI-MSI-edge      virtio0-input.1
 32:          6          0          5          5         26         17   PCI-MSI-edge      virtio0-output.1
 33:    3171188          0          0          0          0       7550   PCI-MSI-edge      virtio0-input.2
 34:          4          5          4          6         17         20   PCI-MSI-edge      virtio0-output.2
 35:         94    1341497          0    1855482          0          0   PCI-MSI-edge      virtio0-input.3
 36:          4          4          7          2         25         16   PCI-MSI-edge      virtio0-output.3
 37:         83         21        354      80506    2820014        117   PCI-MSI-edge      virtio0-input.4
 38:          4          5          3          4         17         16   PCI-MSI-edge      virtio0-output.4
 39:         83          0          0          0          0    2975465   PCI-MSI-edge      virtio0-input.5
 40:          5          1          5          5         23         13   PCI-MSI-edge      virtio0-output.5
 41:        314          0          0          0        857          0   PCI-MSI-edge      snd_hda_intel:card0
NMI:          0          0          0          0          0          0   Non-maskable interrupts
LOC:   12092643   12502450   12556146   12497173   11983902   11983164   Local timer interrupts
SPU:          0          0          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0          0          0   Performance monitoring interrupts
IWI:      30713      35088      30671      35506      27172      28516   IRQ work interrupts
RTR:          0          0          0          0          0          0   APIC ICR read retries
RES:    7486253    6999215    7187159    6896477    6723629    6627373   Rescheduling interrupts
CAL:      13707      17640      17916      14834      15803      13276   Function call interrupts
TLB:     311424     319611     317578     308318     310349     307691   TLB shootdowns
TRM:          0          0          0          0          0          0   Thermal event interrupts
THR:          0          0          0          0          0          0   Threshold APIC interrupts
DFR:          0          0          0          0          0          0   Deferred Error APIC interrupts
MCE:          0          0          0          0          0          0   Machine check exceptions
MCP:         34         34         34         34         34         34   Machine check polls
ERR:          0
MIS:          0
PIN:          0          0          0          0          0          0   Posted-interrupt notification event
NPI:          0          0          0          0          0          0   Nested posted-interrupt event
PIW:          0          0          0          0          0          0   Posted-interrupt wakeup event
```

虽然从这张图里，我们可以看到整体的中断数据，也可以从白底黑字的数据上看到数据的变化，但是我们还没有结论。

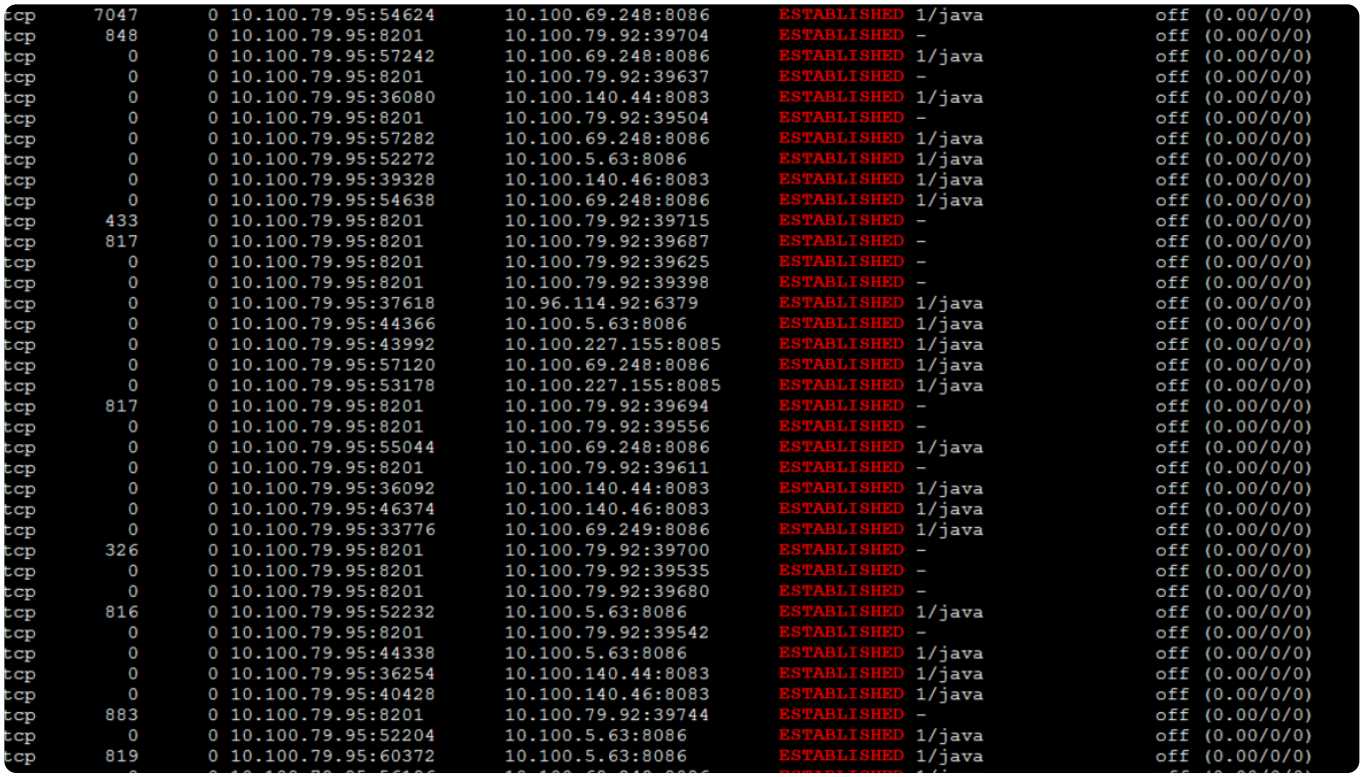
我们再看软中断的数据：

```
Every 5.0s: cat /proc/softirqs

                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
          HI:          7          0          3          0          0         18
       TIMER:    5458391    5662844    5587833    5689075    5084055    5203047
     NET_TX:          4          6         12         13         20         19
     NET_RX:   27170619   27915182   27788736   28177510   24934067   25462099
       BLOCK:      12408      15261      15767      16156      17055      14210
BLOCK_IOPOLL:          0          0          0          0          0          0
     TASKLET:        406        216        311        271        347        319
       SCHED:    1571839    1443795    1355095    1344292    1309496    1293626
     HRTIMER:          0          0          0          0          0          0
         RCU:    3611196    3701890    3662999    3729482    3481010    3588216
```

从白底黑字的数据可以看到，NET_RX 的变化较大（TIMER 是系统的时钟，我们不用做分析），而这个服务器 worker-4 上同时放了 Gateway 和 Redis，并且这两个服务都是用网络的大户，显然这是我们要分析的地方。

由于网络的中断比较高，我们进入到 Pod 中，查看一下网络队列：



```
tcp     7047      0 10.100.79.95:54624      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp      848      0 10.100.79.95:8201       10.100.79.92:39704      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:57242      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39637      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:36080      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39504      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:57282      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:52272      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:39328      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:54638      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp      433      0 10.100.79.95:8201       10.100.79.92:39715      ESTABLISHED -                   off (0.00/0/0)
tcp      817      0 10.100.79.95:8201       10.100.79.92:39687      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39625      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39398      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:37618      10.96.114.92:6379       ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:44366      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:43992      10.100.227.155:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:57120      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:53178      10.100.227.155:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp      817      0 10.100.79.95:8201       10.100.79.92:39694      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39556      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:55044      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39611      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:36092      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:46374      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:33776      10.100.69.249:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp      326      0 10.100.79.95:8201       10.100.79.92:39700      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39535      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39680      ESTABLISHED -                   off (0.00/0/0)
tcp      816      0 10.100.79.95:52232      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:39542      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:44338      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:36254      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:40428      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp      883      0 10.100.79.95:8201       10.100.79.92:39744      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:52204      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp      819      0 10.100.79.95:60372      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
```

你看，这里面有 recv_Q。我们知道，recv_Q 是网络数据的接收队列，它持续有值说明了接收队列中确实有阻塞。

请你注意，这个结论不是我只刷一次 netstat 得到的，而是刷了很多次。因为每次都有这样的队列出现，所以，我才会判断网络接收队列 recv_Q 中确实有未处理完的数据。如果它只是偶尔出现一次，那问题倒是不大。

现在，我们继续分析这个问题，待会再给出解决方案。

既然接收队列中有值，从数据传递的逻辑来看，这应该是上层的应用没有及时处理。因此，我们来跟踪一下方法的执行时间。

在这里，我们先跟踪负责生成确认订单的 generateConfirmOrder 接口（其实这里要跟踪哪个方法倒是无所谓，因为前面说了所有业务都很慢）：

复制代码

```
Command execution times exceed limit: 5, so command will exit. You can set it
Condition express: 1==1 , result: true
`---ts=2021-02-18 19:20:15;thread_name=http-nio-8086-exec-113;id=3528;is_daemo
    `---[151.845221ms] com.dunshan.mall.order.service.impl.PortalOrderServiceI
        `---[151.772564ms] org.springframework.cglib.proxy.MethodInterceptor:i
            `---[151.728833ms] com.dunshan.mall.order.service.impl.PortalOrder
                +---[0.015801ms] com.dunshan.mall.order.domain.ConfirmOrderRes
```

```
                +---[75.263121ms] com.dunshan.mall.order.feign.MemberService:g
                +---[0.006396ms] com.dunshan.mall.model.UmsMember:getId() #9
                +---[0.004322ms] com.dunshan.mall.model.UmsMember:getId() #9
                +---[0.008234ms] java.util.List:toArray() #5
                +---[min=0.006794ms,max=0.012615ms,total=0.019409ms,count=2] o
                +---[0.005043ms] com.dunshan.mall.model.UmsMember:getId() #9
                +---[28.805315ms] com.dunshan.mall.order.feign.CartItemService
                +---[0.007123ms] com.dunshan.mall.order.domain.ConfirmOrderRes
                +---[0.012758ms] com.dunshan.mall.model.UmsMember:getList() #1
                +---[0.011984ms] com.dunshan.mall.order.domain.ConfirmOrderRes
                +---[0.03736ms] com.alibaba.fastjson.JSON:toJSON() #11
                +---[0.010188ms] com.dunshan.mall.order.domain.OmsCartItemVo:<
                +---[0.005661ms] com.dunshan.mall.order.domain.OmsCartItemVo:s
                +---[19.225703ms] com.dunshan.mall.order.feign.MemberService:l
                +---[0.010474ms] com.dunshan.mall.order.domain.ConfirmOrderRes
                +---[0.007807ms] com.dunshan.mall.model.UmsMember:getIntegrati
                +---[0.009189ms] com.dunshan.mall.order.domain.ConfirmOrderRes
                +---[27.471129ms] com.dunshan.mall.mapper.UmsIntegrationConsum
                +---[0.019764ms] com.dunshan.mall.order.domain.ConfirmOrderRes
                +---[0.154893ms] com.dunshan.mall.order.service.impl.PortalOrd
                `---[0.013139ms] com.dunshan.mall.order.domain.ConfirmOrderRes
```

你看，这个接口中有一个 getCurrentMember 方法，它是 Member 上的一个服务，是用来获取当前用户信息的，而其他服务都会用到这个服务，因为需要 Token 嘛。

从上面的栈信息看，getCurrentMember 用了 75ms 多，这个时间明显是慢了，我们跟踪一下这个方法，看看是哪里慢了：

复制代码

```
Condition express: 1==1 , result: true
`---ts=2021-02-18 19:43:18;thread_name=http-nio-8083-exec-25;id=34bd;is_daemon
    `---[36.139809ms] com.dunshan.mall.member.service.imp.MemberServiceImpl:ge
        +---[0.093398ms] javax.servlet.http.HttpServletRequest:getHeader() #18
        +---[0.020236ms] cn.hutool.core.util.StrUtil:isEmpty() #18
        +---[0.147621ms] cn.hutool.json.JSONUtil:toBean() #19
        +---[0.02041ms] com.dunshan.mall.common.domain.UserDto:getId() #19
        `---[35.686266ms] com.dunshan.mall.member.service.MemberCacheService:g
```

这种需要瞬间抓的数据，要反复抓很多遍才能确定。虽然我在这里只展示了一条，但是，我抓的时候可是抓了好多次才得到的。从上面的数据来看，getCurrentMember 中使用的 getMember 方法耗时比较长，达到了 35ms 多。

我们看一下 getMember 的具体实现：

复制代码

```
@Override
public UmsMember getMember(Long memberId) {
    String key = REDIS_DATABASE + ":" + REDIS_KEY_MEMBER + ":" + memberId;
    return (UmsMember) redisService.get(key);

```

这个代码的逻辑很简单：拼接 Key 信息，然后从 Redis 里找到相应的 Member 信息。

既然 getMember 函数是从 Redis 里获取数据，那我们就到 Redis 里检查一下 slowlog：

复制代码

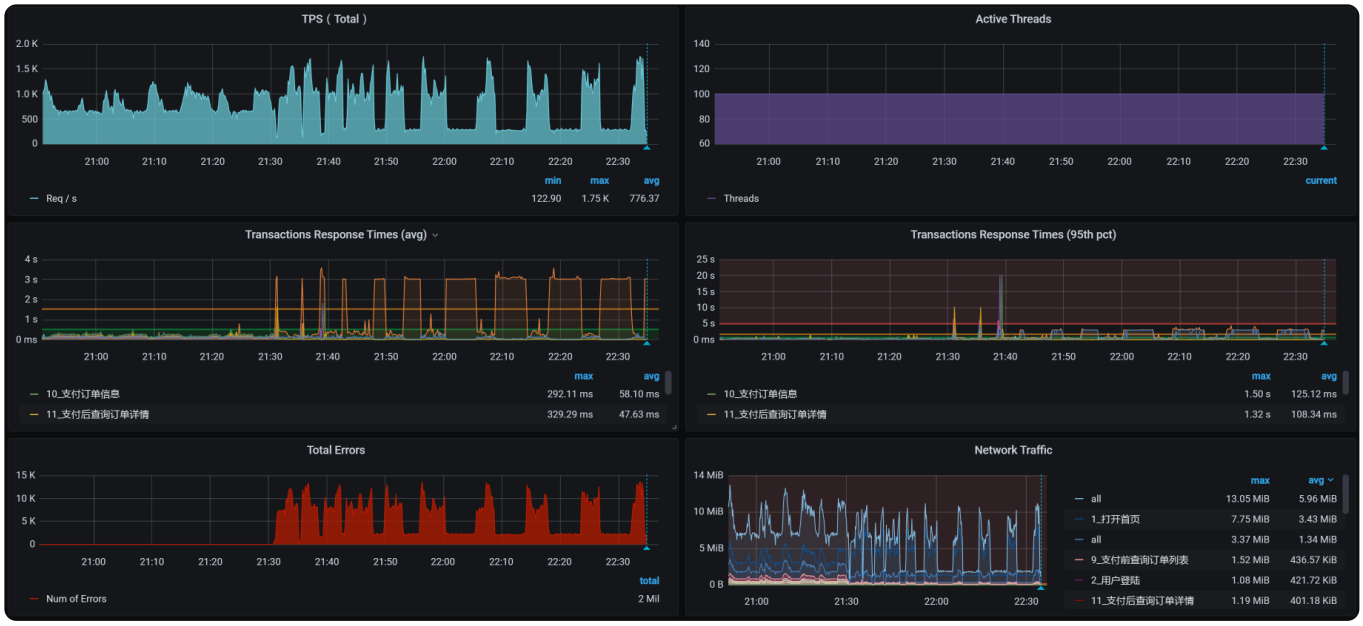
```
127.0.0.1:6379> slowlog get
1) 1) (integer) 5
   1) (integer) 1613647620
   2) (integer) 30577
   3) 1) "GET"
      1) "mall:ums:member:2070064"
   4) "10.100.140.46:53152"
   5) ""
2) 1) (integer) 4
   1) (integer) 1613647541
   2) (integer) 32878
   3) 1) "GET"
      1) "mall:ums:member:955622"
   4) "10.100.140.46:53152"
   5) ""
.......................
```

你看，确实是 get 命令慢了，看时间都超过了 10ms（slowlog 默认设置是 10ms 以上才记录）。如果这个命令执行的次数不多，倒也没啥。关键是这个命令是验证用户的时候用的，这样的时间是没办法容忍的。

为什么这么说呢？

因为在业务上来看，除了打开首页和查询商品不用它之外，其他的脚本似乎都需要用它。所以，它慢了不是影响一个业务，而是影响一堆业务。

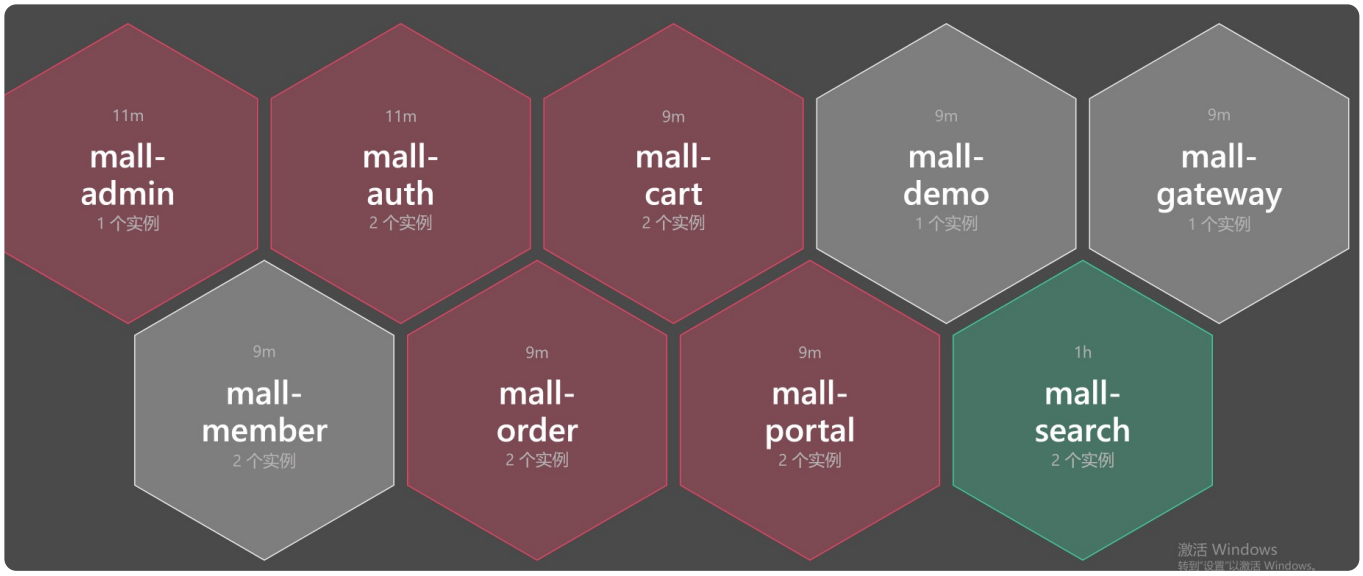
然而，正当我们分析到这里，还没来得及做优化的时候，又..... 出现了新问题。我在接着压的过程中，发现了这样的现象：



你瞧瞧，TPS 不稳定就算了，后面居然还全报错了，这也太不合适了吧！

于是，我开始对报错日志一通查，最后发现了 Redis 的容器都飘红了，下面是 Redis 在架构中的状态截图：

这明显是 Redis 没了呀！这时候我们再去看应用的状态：



满目疮痍呀！

接着，我们登录到 Redis 服务所在的 worker 节点，查看日志：

复制代码

```
[ 7490.807349] redis-server invoked oom-killer: gfp_mask=0xd0, order=0, oom_sc
[ 7490.821216] redis-server cpuset=docker-18cc9a81d8a58856ecf5fed45d7db431885b
[ 7490.826286] CPU: 2 PID: 27225 Comm: redis-server Kdump: loaded Tainted: G
[ 7490.832929] Hardware name: Red Hat KVM, BIOS 0.5.1 01/01/2011
[ 7490.836240] Call Trace:
[ 7490.838006]  [<ffffffff9af7ff85>] dump_stack+0x19/0x1b
[ 7490.841975]  [<ffffffff9af7a8a3>] dump_header+0x90/0x229
[ 7490.844690]  [<ffffffff9aa9c4a8>] ? ep_poll_callback+0xf8/0x220
[ 7490.847625]  [<ffffffff9a9c246e>] oom_kill_process+0x25e/0x3f0
[ 7490.850515]  [<ffffffff9a933a41>] ? cpuset_mems_allowed_intersects+0x21/0x3
[ 7490.853893]  [<ffffffff9aa40ba6>] mem_cgroup_oom_synchronize+0x546/0x570
[ 7490.857075]  [<ffffffff9aa40020>] ? mem_cgroup_charge_common+0xc0/0xc0
[ 7490.860348]  [<ffffffff9a9c2d14>] pagefault_out_of_memory+0x14/0x90
[ 7490.863651]  [<ffffffff9af78db3>] mm_fault_error+0x6a/0x157
[ 7490.865928]  [<ffffffff9af8d8d1>] __do_page_fault+0x491/0x500
[ 7490.868661]  [<ffffffff9af8da26>] trace_do_page_fault+0x56/0x150
[ 7490.871811]  [<ffffffff9af8cfa2>] do_async_page_fault+0x22/0xf0
[ 7490.874423]  [<ffffffff9af897a8>] async_page_fault+0x28/0x30
[ 7490.877127] Task in /kubepods.slice/kubepods-burstable.slice/kubepods-burst
[ 7490.893825] memory: usage 3145728kB, limit 3145728kB, failcnt 176035
[ 7490.896099] memory+swap: usage 3145728kB, limit 3145728kB, failcnt 0
[ 7490.899137] kmem: usage 0kB, limit 9007199254740988kB, failcnt 0
[ 7490.902012] Memory cgroup stats for /kubepods.slice/kubepods-burstable.slic
[ 7490.962494] [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_scor
[ 7490.966577] [27197]     0 27197      596      166       5        0
[ 7490.970286] [27225]     0 27225   818112   786623    1550        0
[ 7490.974006] [28322]     0 28322      999      304       6        0
[ 7490.978178] Memory cgroup out of memory: Kill process 27242 (redis-server)
[ 7490.983765] Killed process 27225 (redis-server), UID 0, total-vm:3272448kB,
```

原来是 worker 节点的内存不够用了，而 Redis 在计算 OOM 评分时也达到了 1808 分。于是，操作系统就义无反顾地把 Redis 给杀了。

我们再次把 Redis 启动之后，观察它的内存消耗，结果如下：

复制代码

```
[root@k8s-worker-4 ~]# pidstat -r -p 5356 1
Linux 3.10.0-1127.el7.x86_64 (k8s-worker-4)   2021年02月18日   _x86_64_  (6 CPU

19时55分52秒   UID       PID  minflt/s  majflt/s     VSZ    RSS   %MEM  Command
19时55分53秒     0      5356     32.00      0.00 3272448 1122152   6.90  redis-
19时55分54秒     0      5356     27.00      0.00 3272448 1122416   6.90  redis-
19时55分55秒     0      5356     28.00      0.00 3272448 1122416   6.90  redis-
19时55分56秒     0      5356     28.00      0.00 3272448 1122680   6.90  redis-
19时55分57秒     0      5356     21.78      0.00 3272448 1122680   6.90  redis-
19时55分58秒     0      5356     38.00      0.00 3272448 1122880   6.90  redis-
19时55分59秒     0      5356     21.00      0.00 3272448 1122880   6.90  redis-
19时56分00秒     0      5356     25.00      0.00 3272448 1122880   6.90  redis-
```

我只是截取了 Redis 没死之前的一小段数据，然后通过 RSS（实际使用内存）来不断观察这段数据，发现内存确实会一直往上涨。我又查了一下 Redis 的配置文件，发现没配置 maxmemory。

没配置倒是没什么，内存不够就不够了呗，Pod 不是还有内存限制吗？但可惜的是，worker 上的内存不够了，导致了 Redis 进程被操作系统杀掉了，这就解释了 TPS 图中后半段会报错的问题。

但是响应时间慢，我们还是得接着分析。我们在前面看到软中断和带宽有关，为了减少服务中断之间的相互影响，待会我把 Redis 和 Gateway 两个服务分开。

我们都知道，Redis 是靠内存来维护数据的，如果只做内存的操作，它倒是会很快。但是 Redis 还有一块跟内存比较有关的功能，就是持久化。我们现在采用的是 AOF 持久化策略，并且没有限制 AOF 的文件大小。

这个持久化文件是放到 NFS 文件服务器上面的，既然是放到文件服务器上，那就需要有足够的磁盘 IO 能力才可以。因此，我们到 nfs 服务器上查看一下 IO 的能力，截取部分数据如下：

```
Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00   65.00    0.00  6516.00     0.00   200.49

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00   24.00    0.00   384.00     0.00    32.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00    8.00    0.00  1124.00     0.00   281.00

.........................

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00   11.00    0.00   556.00     0.00   101.09

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00    4.00    0.00    32.00     0.00    16.00
```

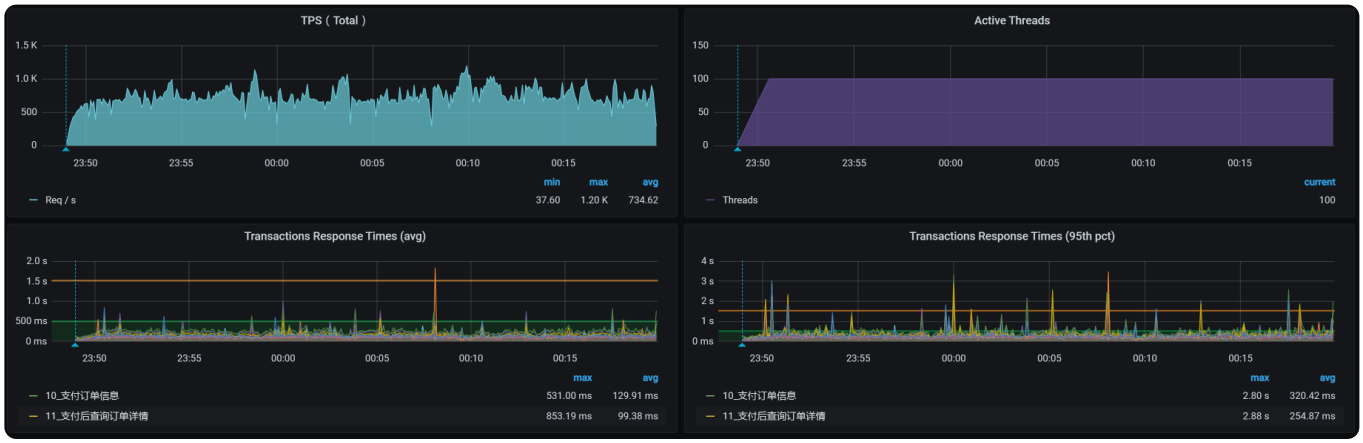
通过 svctm（IO 响应时间计数器）这个参数可以看到，IO 的响应时间也增加了。虽然在 sysstat 的新版本中已经不建议使用 svctm 了，但是在我们当前使用的版本中，仍然有这个参数。并且通过它，我们可以看到 IO 的响应时间确实在增加。

为了证明 IO 的响应时间是和 AOF 有关，我们先把 AOF 关掉，设置 appendonly no 看看效果。如果有效果，那优化方向就非常明确了，我们要做的就是这几个优化动作：

1. 把 Redis 先移到一个网络需求没那么大的 Worker 上去，观察一下 TPS 能不能好一点。如果这一步有效果，我们就不用再折腾下一步了；
2. 如果上一步做完之后没有效果，就再把 AOF 关掉，再观察 TPS。如果 AOF 关掉后有效果，那我们就得分析下这个应用有没有必要做 Redis 的持久化了。如果有必要，就得换个快一点的硬盘；

3. 不管上面两步有用没用，对 Redis 来说，我们都应该考虑限制内存的大小和 AOF 文件的大小。

我们看一下把 Redis 从 worker-4 移到 worker-7 上之后的 TPS 如下：



TPS 还是在下降，并且没有一开始的那么高了，之前都能达到 1000TPS。这个看似非常正确的优化动作却导致了 TPS 下降的现象，显然不是我们期望的。

现在还不知道问题在哪里，不过，我们一直想达到的目标是降队列。所以，我们先确认下网络队列有没有降下来，再来考虑 TPS 怎么提升。

你看 worker-4 上的队列，没有 recv_Q 的值了：

```
tcp        0      0 10.100.79.95:8201       10.100.79.92:42410      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:33788      10.100.227.156:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:58248      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:36578      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:34184      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42242      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42550      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:58346      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:36642      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42234      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:50128      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:33790      10.100.227.156:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:54288      10.100.5.63:8086        ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:56952      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42338      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:44818      10.100.69.249:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:40420      10.100.69.249:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42589      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:33528      10.100.227.156:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:58426      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:34954      10.100.69.249:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:57056      10.100.227.156:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:56814      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:56856      10.100.140.44:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:50228      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42371      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:35190      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:50206      10.100.140.46:8083      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42484      ESTABLISHED -                   off (0.00/0/0)
tcp        0      0 10.100.79.95:36622      10.100.69.248:8086      ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:33890      10.100.227.156:8085     ESTABLISHED 1/java              off (0.00/0/0)
tcp        0      0 10.100.79.95:8201       10.100.79.92:42544      ESTABLISHED -                   off (0.00/0/0)
```

现在我们就得来处理下 AOF 了，因为我们虽然移开了 Redis，但是 TPS 并没有上升。所以，我们还得看看 AOF 的影响。

关掉 AOF 之后，TPS 如下：

总体资源如下：

IP (链接到明细)	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	2.02 week	7.64 GiB	4	2.44	39.03%	41.98%	29.95%	9.21 kB/s	94.19 kB/s	308	97	415.89 kbps	572.25 kbps
k8s-master-2	k8s-master-2	8.72 week	7.64 GiB	4	0.51	21.03%	23.91%	29.95%	0 B/s	95.66 kB/s	255	31	239.98 kbps	243.63 kbps
k8s-master-3	k8s-master-3	2.02 week	7.64 GiB	4	2.91	37.15%	36.65%	29.95%	23.02 kB/s	96.44 kB/s	305	96	542.77 kbps	582.10 kbps
k8s-worker-1	k8s-worker-1	22.27 hour	15.51 GiB	6	5.47	55.91%	68.17%	51.33%	231.44 kB/s	2.78 MB/s	29	69	33.52 Mbps	33.63 Mbps
k8s-worker-2	k8s-worker-2	1.39 day	15.51 GiB	6	20.73	72.44%	38.41%	37.60%	23.72 kB/s	83.98 kB/s	28	648	13.27 Mbps	9.47 Mbps
k8s-worker-3	k8s-worker-3	1.39 day	15.51 GiB	6	13.09	65.27%	28.37%	41.30%	13.81 kB/s	73.88 kB/s	28	418	14.44 Mbps	16.35 Mbps
k8s-worker-4	k8s-worker-4	1.39 day	15.51 GiB	6	15.94	82.09%	41.47%	46.81%	9.25 kB/s	243.87 kB/s	33	21 K	22.69 Mbps	26.15 Mbps
k8s-worker-5	k8s-worker-5	7.63 week	15.51 GiB	6	3.43	39.71%	58.89%	38.45%	23.02 kB/s	17.18 kB/s	28	150	14.59 Mbps	14.24 Mbps
k8s-worker-6	k8s-worker-6	7.63 week	15.51 GiB	6	9.46	56.99%	34.68%	42.45%	13.81 kB/s	47.93 kB/s	47	281	17.58 Mbps	13.24 Mbps
k8s-worker-7	k8s-worker-7	7.33 week	15.51 GiB	4	1.57	33.34%	33.48%	44.50%	23.02 kB/s	131.38 kB/s	30	283	11.07 Mbps	10.93 Mbps
k8s-worker-8	k8s-worker-8	6.15 week	15.51 GiB	4	0.47	12.26%	33.75%	57.35%	0 B/s	4.39 kB/s	28	203	16.81 Mbps	16.90 Mbps
k8s-worker-9	k8s-worker-9	7.33 week	15.51 GiB	6	1.83	23.91%	28.61%	29.95%	4.60 kB/s	9.42 kB/s	28	79	27.34 Mbps	37.02 Mbps

看到了没，效果还是有的吧？我们可是得到了 1000 以上的稳定的 TPS 曲线。

在这个容量场景中，我们完成了四个阶段的分析之后，优化效果还不错。不过，每个性能测试都应该有**结论**。所以，我们还需要做一个动作，就是接着增加压力，看一下系统的最大容量能达到多少。

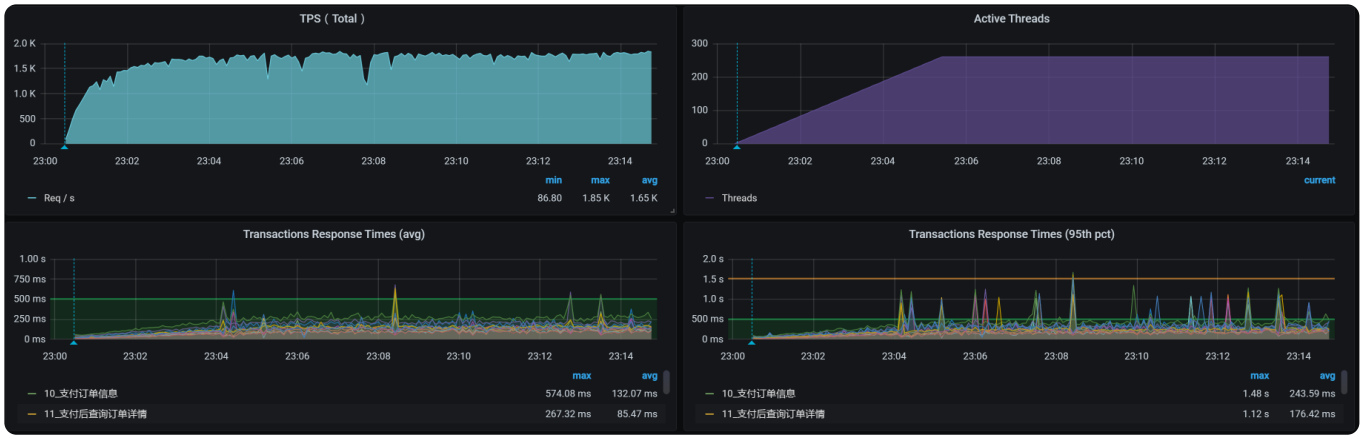
于是，我们进入第五个阶段的分析。

第五阶段分析

请你注意，**容量场景最重要的变化只有一个，就是增加线程**。而跟着线程一起变化的就是参数化的数据量。在这样的增加线程的场景中，我们还要关注的就是资源的均衡使用。因此，在第四阶段的优化之后，我们先来看一下这个场景的结果是个什么样子。

场景运行数据

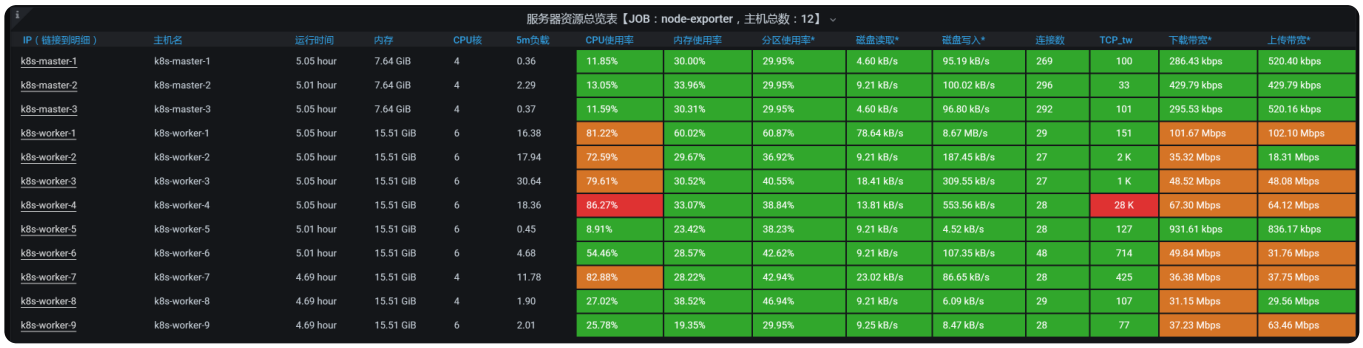
场景压力数据如下：



从效果上来看，不错哦，TPS 已经达到 1700 了。

全局监控分析

全局监控的数据如下：



IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	5.05 hour	7.64 GiB	4	0.36	11.85%	30.00%	29.95%	4.60 kB/s	95.19 kB/s	269	100	286.43 kbps	520.40 kbps
k8s-master-2	k8s-master-2	5.01 hour	7.64 GiB	4	2.29	13.05%	33.96%	29.95%	9.21 kB/s	100.02 kB/s	296	33	429.79 kbps	429.79 kbps
k8s-master-3	k8s-master-3	5.05 hour	7.64 GiB	4	0.37	11.59%	30.31%	29.95%	4.60 kB/s	96.80 kB/s	292	101	295.53 kbps	520.16 kbps
k8s-worker-1	k8s-worker-1	5.05 hour	15.51 GiB	6	16.38	81.22%	60.02%	60.87%	78.64 kB/s	8.67 MB/s	29	151	101.67 Mbps	102.10 Mbps
k8s-worker-2	k8s-worker-2	5.05 hour	15.51 GiB	6	17.94	72.59%	29.67%	36.92%	9.21 kB/s	187.45 kB/s	27	2 K	35.32 Mbps	18.31 Mbps
k8s-worker-3	k8s-worker-3	5.05 hour	15.51 GiB	6	30.64	79.61%	30.52%	40.55%	18.41 kB/s	309.55 kB/s	27	1 K	48.52 Mbps	48.08 Mbps
k8s-worker-4	k8s-worker-4	5.05 hour	15.51 GiB	6	18.36	86.27%	33.07%	38.84%	13.81 kB/s	553.56 kB/s	28	28 K	67.30 Mbps	64.12 Mbps
k8s-worker-5	k8s-worker-5	5.01 hour	15.51 GiB	6	0.45	8.91%	23.42%	38.23%	9.21 kB/s	4.52 kB/s	28	127	931.61 kbps	836.17 kbps
k8s-worker-6	k8s-worker-6	5.01 hour	15.51 GiB	6	4.68	54.46%	28.57%	42.62%	9.21 kB/s	107.35 kB/s	48	714	49.84 Mbps	31.76 Mbps
k8s-worker-7	k8s-worker-7	4.69 hour	15.51 GiB	4	11.78	82.88%	28.22%	42.94%	23.02 kB/s	86.65 kB/s	28	425	36.38 Mbps	37.75 Mbps
k8s-worker-8	k8s-worker-8	4.69 hour	15.51 GiB	4	1.90	27.02%	38.52%	46.94%	9.21 kB/s	6.09 kB/s	29	107	31.15 Mbps	29.56 Mbps
k8s-worker-9	k8s-worker-9	4.69 hour	15.51 GiB	6	2.01	25.78%	19.35%	29.95%	9.25 kB/s	8.47 kB/s	28	77	37.23 Mbps	63.46 Mbps

从上面两张图中可以看到，在我们这样的压力之下，TPS 最大能达到 1700 左右，系统整体资源使用率也不算少了。

经过了基准场景和容量场景之后，我们现在就可以下结论了：系统资源在这个最大容量的场景中已经达到了比较高的使用率。

你有没有听过性能行业中一直流传的一句话：性能优化是无止境的。所以，**我们一定要选择好性能项目结束的关键点。**

就拿我们这个课程的案例来说，这个系统在技术上已经没有优化的空间了，或者说在技术上优化的成本比较高（比如说要做一些定制的开发和改造）。如果你在这种情况下还想要扩大容量，那么你能做的就是增加节点和硬件资源，把所有的硬件资源全都用完。

但是！请所有做性能项目的人注意！**我们做性能项目，不是说把系统优化到最好后，就可以在生产环境中按这样的容量来设计整体的生产资源 **** 了**！要知道，生产环境中出现问题的代价是非常高的，一般我们都会增加一定的冗余，但是冗余多少就不一而足了。

在很多企业中，生产环境里使用的 CPU 都没有超过 20%。为什么会有这么多的冗余呢？在我的经验中，大部分的项目都是根据业务的发展在不断迭代，然后形成了这样的局面。你可以想像一下，这样的生产环境里有多少资源浪费。

说到这里，我们不得不说一下怎么评估架构级的容量。因为对于一个固定客户端数的系统来说，很容易判断整体的容量。但是，对非固定客户端数的系统而言，要想抵挡得住突发的业务容量。那就要经过严格的设计了，像缓存、队列、限流、熔断、预热等等这些手段都要上了。

对于整体的架构容量设计，在所有的企业中都不是一蹴而就的，都要经过多次的、多年的版本迭代，跟着业务的发展不断演进得到。这就不是一个专栏可以尽述的了。

总结

从基准场景做完之后，我们来到了容量场景，这是一个非常大的变化。在这个场景中，我们解决了几个问题并最终给出了结论：

第一个阶段：分析了压力工具参数化的问题，解决了 TPS 不断降低、响应时间不断上升的问题。

第二个阶段：分析了数据库索引，解决了 TPS 低的问题。

第三个阶段：分析了资源争用，解决了多容器跑到一个节点上的问题。

第四个阶段：分析了网络争用和 Redis 的 AOF，解决了 TPS 不稳定的问题。

第五个阶段：递增压力，给出最终系统整体容量的结论。

在做完这些动作之后，我们终于可以给出比较明确的结论了：TPS 能达到 1700！

请你记住，对于一个性能项目来说，没有结论就是在耍流氓。所以，我一直强调，**在性能项目中，我们一定要给出最大容量的结论。**

课后作业

这就是今天的全部内容，最后给你留两个思考题吧：

1. 为什么性能项目一定要有结论？
2. 当多个性能问题同时出现时，我们怎么判断它们产生的相互影响？
3. 如何判断一个系统已经优化到了最优的状态？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果这节课让你有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 容量场景之一：索引优化和Kubernetes资源分配不均衡怎么办？

下一篇 26 | 稳定性场景之一：怎样搞定业务积累量产生的瓶颈问题？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。